

06 | auto/decltype：为什么要有自动类型推导？

2020-05-19 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 12:39 大小 11.60M



你好，我是 Chrono。

前两周我们从宏观的层面上重新认识了 C++，从今天开始，我们将进入一个新的“语言特性”单元，“下沉”到微观的层面去观察 C++，一起去见一些老朋友、新面孔，比如 const、exception、lambda。

这次要说的，就是 C++11 里引入的一个很重要的语言特性：自动类型推导。



自动类型推导

如果你有过一些 C++ 的编程经验，了解过 C++11，那就一定听说过“**自动类型推导**”（auto type deduction）。

它其实是一个非常“老”的特性，C++ 之父 Bjarne Stroustrup (B·S) 早在 C++ 诞生之初就设计并实现了它，但因为与早期 C 语言的语义有冲突，所以被“雪藏”了近三十年。直到 C99 消除了兼容性问题，C++11 才让它再度登场亮相。

那为什么要重新引入这个“老特性”呢？为什么非要有“自动类型推导”呢？

我觉得，你可以先从字面上去理解，把这个词分解成三个部分：“自动”“类型”和“推导”。

“自动”就是让计算机去做，而不是人去做，相对的是“手动”。

“类型”指的是操作目标，出来的是编译阶段的类型，而不是数值。

“推导”就是演算、运算，把隐含的值给算出来。

好，我们来看一看“自动类型推导”之外的其他几种排列组合，通过对比的方式来帮你理解它。

像计算“ $a = 1 + 1$ ”，你可以在写代码的时候直接填上 2，这就是“手动数值推导”。你也可以“偷懒”，只写上表达式，让电脑在运行时自己算，这就是“自动数值推导”。

“数值推导”对于人和计算机来说都不算什么难事，所以手动和自动的区别不大，只有快慢的差异。但“类型推导”就不同了。

因为 C++ 是一种静态强类型的语言，任何变量都要有一个确定的类型，否则就不能用。在“自动类型推导”出现之前，我们写代码时只能“手动推导”，也就是说，在声明变量的时候，必须要明确地给出类型。

这在变量类型简单的时候还好说，比如 int、double，但在泛型编程的时候，麻烦就来了。因为泛型编程里会有很多模板参数，有的类型还有内部子类型，一下子就把 C++ 原本简洁的类型体系给搞复杂了，这就迫使我们去和编译器“斗智斗勇”，只有写对了类型，编译器才会“放行”（编译通过）。

```

1  int      i = 0;           // 整数变量，类型很容易知道
2  double   x = 1.0;        // 浮点数变量，类型很容易知道
3
4  std::string str = "hello"; // 字符串变量，有了名字空间，麻烦了一点
5
6  std::map<int, std::string> m = // 关联数组，名字空间加模板参数，很麻烦
7      {{1,"a"}, {2,"b"}};      // 使用初始化列表的形式
8
9  std::map<int, std::string>::const_iterator // 内部子类型，超级麻烦
10 iter = m.begin();
11
12 ??? = bind1st(std::less<int>(), 2); // 根本写不出来

```

虽然你可以用 `typedef` 或者 `using` 来简化类型名，部分减轻打字的负担，但关键的“手动推导”问题还是没有得到解决，还是要去翻看类型定义，找到正确的声明。这时，C++ 的静态强类型的优势反而成为了劣势，阻碍了程序员的工作，降低了开发效率。

其实编译器是知道（而且也必须知道）这些类型的，但它却没有办法直接告诉你，这就很尴尬了。一边是急切地想知道答案，而另一边却只给判个对错，至于怎么错了、什么是正确答案，“打死了也不说”。

但有了“自动类型推导”，问题就迎刃而解了。这就像是在编译器紧闭的大门上开了道小口子，你跟它说一声，它就递过来张小纸条，具体是什么不重要，重要的是里面存了我们想要的类型。

这个“小口子”就是关键字 **auto**，在代码里的作用像是个“占位符”（placeholder）。写上它，你就可以让编译器去自动“填上”正确的类型，既省力又省心。

```

1  auto i = 0;           // 自动推导为int类型
2  auto x = 1.0;        // 自动推导为double类型
3
4  auto str = "hello";  // 自动推导为const char [6]类型
5
6  std::map<int, std::string> m = {{1,"a"}, {2,"b"}}; // 自动推导不出来
7
8  auto iter = m.begin(); // 自动推导为map内部的迭代器类型
9
10 auto f = bind1st(std::less<int>(), 2); // 自动推导出类型，具体是啥不知道

```

不过需要注意的是，因为 C++ 太复杂，“自动类型推导”有时候可能失效，给不出你想要的结果。比如，在上面的这段代码里，就把字符串的类型推导成了“const char [6]”而不是“std::string”。而有的时候，编译器也理解不了代码的意思，推导不出恰当的类型，还得你自己“亲力亲为”。

在这个示例里，你还可以直观感觉到 auto 让代码干净整齐了很多，不用去写那些复杂的模板参数了。但如果你把“自动类型推导”理解为仅仅是简化代码、少打几个字，那就实在是浪费了 C++ 标准委员会的一番苦心。

除了简化代码，auto 还避免了对类型的“硬编码”，也就是说变量类型不是“写死”的，而是能够“自动”适应表达式的类型。比如，你把 map 改为 unordered_map，那么后面的代码都不用动。这个效果和类型别名（[第 5 讲](#)）有点像，但你不需要写出 typedef 或者 using，全由 auto “代劳”。

另外，你还应该认识到，“自动类型推导”实际上和“attribute”一样（[第 4 讲](#)），是编译阶段的特殊指令，指示编译器去计算类型。所以，它在泛型编程和模板元编程里还有更多的用处，后面我会陆续讲到。

认识 auto

刚才说了，auto 有时候会不如你设想的那样工作，因此在使用的时候，有一些需要特别注意的地方，下面我就给你捋一捋。

首先，你要知道，auto 的“自动推导”能力只能用在“**初始化**”的场合。

具体来说，就是**赋值初始化**或者**花括号初始化**（初始化列表、Initializer list），变量右边必须要有一个表达式（简单、复杂都可以）。这样你才能在左边放上 auto，编译器才能找到表达式，帮你自动计算类型。


如果不是初始化的形式，只是“纯”变量声明，那就无法使用 auto。因为这个时候没有表达式可以让 auto 去推导。

 复制代码

```
1 auto x = 0L;      // 自动推导为long
2 auto y = &x;      // 自动推导为long*
3 auto z {&x};      // 自动推导为long*
```

```
4
5 auto err;           // 错误，没有赋值表达式，不知道是什么类型
```

这里还有一个特殊情况，在类成员变量初始化的时候（[第 5 讲](#)），目前的 C++ 标准不允许使用 auto 推导类型（但我个人觉得其实没有必要，也许以后会放开吧）。所以，在类里你还是要老老实实地去“手动推导类型”。

 复制代码

```
1 class X final
2 {
3     auto a = 10; // 错误，类里不能使用auto推导类型
4 };
```

知道了应用场合，你还需要了解 auto 的推导规则，保证它能够按照你的意思去工作。虽然标准里规定得很复杂、很细致，但我总结出了两条简单的规则，基本上够用了：

auto 总是推导出“值类型”，绝不会是“引用”；

auto 可以附加上 const、volatile、*、& 这样的类型修饰符，得到新的类型。

下面我举几个例子，你一看就能明白：

 复制代码

```
1 auto      x = 10L;    // auto推导为long, x是long
2
3 auto&     x1 = x;      // auto推导为long, x1是long&
4 auto*     x2 = &x;     // auto推导为long, x2是long*
5 const auto& x3 = x;    // auto推导为long, x3是const long&
6 auto      x4 = &x3;    // auto推导为const long*, x4是const long*
```

认识 decltype

前面我都在说 auto，其实，C++ 的“自动类型推导”还有另外一个关键字：**decltype**。


刚才你也看到了，auto 只能用于“初始化”，而这种“**向编译器索取类型**”的能力非常有价值，把它限制在这么小的场合，实在是有点“屈才”了。

“自动类型推导” 要求必须从表达式推导，那在没有表达式的时候，该怎么办呢？

其实解决思路也很简单，就是“自己动手，丰衣足食”，自己带上表达式，这样就走到哪里都不怕了。

`decltype` 的形式很像函数，后面的圆括号里就是可用于计算类型的表达式（和 `sizeof` 有点类似），其他方面就和 `auto` 一样了，也能加上 `const`、`*`、`&` 来修饰。

因为它已经自带表达式，所以不需要变量后面再有表达式，也就是说可以直接声明变量。

 复制代码

```
1  int x = 0;           // 整型变量
2
3  decltype(x)          x1;      // 推导为int, x1是int
4  decltype(x)&          x2 = x;  // 推导为int, x2是int&, 引用必须赋值
5  decltype(x)*          x3;      // 推导为int, x3是int*
6  decltype(&x)          x4;      // 推导为int*, x4是int*
7  decltype(&x)*          x5;      // 推导为int*, x5是int**
8  decltype(x2)          x6 = x2; // 推导为int&, x6是int&, 引用必须赋值
```


把 `decltype` 和 `auto` 比较一下，简单来看，好像就是把表达式改到了左边而已，但实际上，在推导规则上，它们有一点细微且重要的区别：

`decltype` 不仅能够推导出值类型，还能够推导出引用类型，也就是表达式的“原始类型”。

在示例代码中，我们可以看到，除了加上 `*` 和 `&` 修饰，`decltype` 还可以直接从一个引用类型的变量推导出引用类型，而 `auto` 就会把引用去掉，推导出值类型。

所以，你完全可以把 `decltype` 看成是一个真正的类型名，用在变量声明、函数参数 / 返回值、模板参数等任何类型能出现的地方，只不过这个类型是在编译阶段通过表达式“计算”得到的。

如果不信的话，你可以用 `using` 类型别名来试一试。

 复制代码

```
using int_ptr = decltype(&x);    // int *
```

既然 `decltype` 类型推导更精确，那是不是可以替代 `auto` 了呢？

实际上，它也有个缺点，就是写起来略麻烦，特别在用于初始化的时候，表达式要重复两次（左边的类型计算，右边的初始化），把简化代码的优势完全给抵消了。

所以，C++14 就又增加了一个 “**`decltype(auto)`**” 的形式，既可以精确推导类型，又能像 `auto` 一样方便使用。

 复制代码

```
1  int x = 0;           // 整型变量
2
3  decltype(auto) x1 = (x); // 推导为int&, 因为(expr)是引用类型
4  decltype(auto) x2 = &x;  // 推导为int*
5  decltype(auto) x3 = x1;  // 推导为int&
```

使用 `auto/decltype`

现在，我已经讲完了 “自动类型推导” 的两个关键字：`auto` 和 `decltype`，那么，该怎么用好它们呢？

我觉得，因为 `auto` 写法简单，推导规则也比较好理解，所以，**在变量声明时应该尽量多用 `auto`**。前面已经举了不少例子，这里就不再重复了。


`auto` 还有一个 “最佳实践”，就是 “**`range-based for`**”，不需要关心容器元素类型、迭代器返回值和首末位置，就能非常轻松地完成遍历操作。不过，为了保证效率，最好使用 “`const auto&`” 或者 “`auto&`”。

 复制代码

```
1  vector<int> v = {2,3,5,7,11}; // vector顺序容器
2
3  for(const auto& i : v) {       // 常引用方式访问元素，避免拷贝代价
4      cout << i << ", ";       // 常引用不会改变元素的值
5  }
6
7  for(auto& i : v) {             // 引用方式访问元素
8      i++;                       // 可以改变元素的值
```

```
9     cout << i << ",";
10 }
```

在 C++14 里，auto 还新增了一个应用场合，就是能够推导函数返回值，这样在写复杂函数的时候，比如返回一个 pair、容器或者迭代器，就会很省事。


 复制代码

```
1 auto get_a_set()           // auto作为函数返回值的占位符
2 {
3     std::set<int> s = {1,2,3};
4     return s;
5 }
```

再来看 decltype 怎么用最合适。


它是 auto 的高级形式，更侧重于编译阶段的类型计算，所以常用在泛型编程里，获取各种类型，配合 typedef 或者 using 会更加方便。当你感觉“这里我需要一个特殊类型”的时候，选它就对了。

比如说，定义函数指针在 C++ 里一直是个比较头疼的问题，因为传统的写法实在是太怪异了。但现在就简单了，你只要手里有一个函数，就可以用 decltype 很容易得到指针类型。

 复制代码

```
1 // UNIX信号函数的原型，看着就让人晕，你能手写出函数指针吗？
2 void (*signal(int signo, void (*func)(int)))(int)
3
4 // 使用decltype可以轻松得到函数指针类型
5 using sig_func_ptr_t = decltype(&signal) ;
```

在定义类的时候，因为 auto 被禁用了，所以这也是 decltype 可以“显身手”的地方。它可以搭配别名任意定义类型，再应用到成员变量、成员函数上，变通地实现 auto 的功能。

 复制代码

```
1 class DemoClass final
2 {
3 public:
4     using set_type      = std::set<int>; // 集合类型别名
```



```
5 private:
6     set_type      m_set;                // 使用别名定义成员变量
7
8     // 使用decltype计算表达式的类型，定义别名
9     using iter_type = decltype(m_set.begin());
10
11    iter_type      m_pos;                // 类型别名定义成员变量
12
```

小结

好了，今天我介绍了 C++ 里的“自动类型推导”，简单小结一下今天的内容。

1. “自动类型推导”是给编译器下的指令，让编译器去计算表达式的类型，然后返回给程序员。
2. auto 用于初始化时的类型推导，总是“值类型”，也可以加上修饰符产生新类型。它的规则比较好理解，用法也简单，应该积极使用。
3. decltype 使用类似函数调用的形式计算表达式的类型，能够用在任意场合，因为它就是一个编译阶段的类型。
4. decltype 能够推导出表达式的精确类型，但写起来比较麻烦，在初始化时可以采用 decltype(auto) 的简化形式。
5. 因为 auto 和 decltype 不是“硬编码”的类型，所以用好它们可以让代码更清晰，减少后期维护的成本。

课下作业

最后是课下作业时间，给你留两个思考题：

1. auto 和 decltype 虽然很方便，但用多了也确实会“隐藏”真正的类型，增加阅读时的理解难度，你觉得这算是缺点吗？是否有办法克服或者缓解？
2. 说一下你对 auto 和 decltype 的认识。你认为，两者有哪些区别呢？（推导规则、应用场合等）

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友，我们下节课见。

课外小贴士

1. 在C语言里，auto关键字最早的含义是表示局部变量，与static同级，但因为用得极少，所以到了C++11，为了避免再新增关键字，就给“变废为宝”了。
2. C++标准又特别规定，类的静态成员变量允许使用auto自动推导类型，但我建议，为了与非静态成员保持一致，还是统一不使用auto比较好。
3. C++14新增了字面量后缀“s”，表示标准字符串，所以就可以用“auto str = "xxx"s;”的形式直接推导出std::string类型。
4. C++17为auto增加了一种叫“结构化绑定”的功能，相当于简化了的tie()用法。

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 05 | 面向对象编程: 怎样才能写出一个“好”的类?

下一篇 07 | const/volatile/mutable: 常量/变量究竟是怎么回事?

精选留言 (16)

写留言



Mervin

2020-05-19

课后题:

1. 给程序作者带来了一些便利, 但是给读者比较大的麻烦, 所以我认为尽量还是应该在比较清晰明确的地方使用, 并加以明确的注释。
2. auto推导的是编译器计算变量初始值得到类型的, decltype也是分析表达式但是不需要计算表达式, 所以它与表达式本身有很大关系。

展开

作者回复:

1. 说的很好, 所以要用好auto还是要掌握一个度。

2.auto和decltype的编译期计算类型过程是一样的，都是得出类型，不会计算表达式，只是一个从初始化里获取表达式，一个自带表达式，这个区别导致了用法的不同。



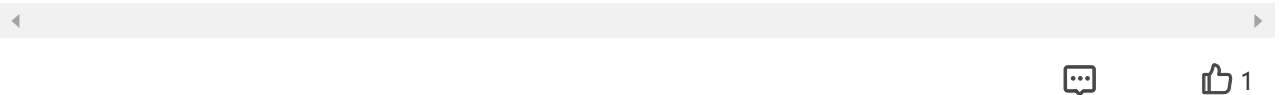
Geek_197dc8

2020-05-21

auto总是推导出“值类型”，但绝不会是“引用”，这句话怎么理解，难道不可以推导出引用的类型嘛？我看你的例子 `auto& x1 = x`，不是推导出引用类型嘛。

展开 ∨

作者回复: 注意细看，auto还是值类型，而“auto&”才是引用类型。



EncodedStar

2020-05-19

我觉得auto 虽然方便了，但是代码不能都用auto吧，大量的auto反而让程序员摸不着头脑，这就像看一本书所有地方都花下划线就失去了下划线的意义。

auto对于减少冗赘的代码也很有用。比如：之前我们写代码是：

```
for(vector<int>::const_iterator itr = m_vector.begin(); itr != m_vector.end(); ++itr)
```

可以使用auto简化为：...

展开 ∨

作者回复: 如果觉得auto太多，可以先试着用它来简化复杂类型的声明，然后再慢慢扩展应用场合。

decltype在泛型编程和模板元编程里非常有用。



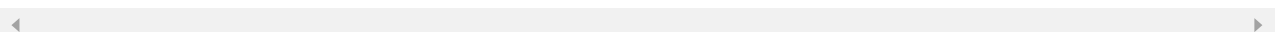
Jason

2020-05-19

"auto 和 decltype 虽然很方便，但用多了也确实会“隐藏”真正的类型，增加阅读时的理解难度。”

我觉得这很算缺点，它们应该只用在确实很难手动推导出变量类型的地方。

作者回复: 对，所以auto不宜用的太多，要适当，最好能够加一下注释，说一下这个auto是什么。



**九三**

2020-05-30

老师，总是在一些c++ 源码看到extern “C” 对这个关键字理解的不是很透

作者回复: 这个是为了兼容C语言，因为C++编译生成的链接符号与C不一样，用这个就会导出与C一样规则的符号，方便外部库调用。

可以再搜一下相关的资料，看几篇就能理解了。

**yelin**

2020-05-27

2.

推导规则: auto值总是值类型, decltype 不仅能够推导出值类型, 还能够推导出引用类型。&,* ,const的属性也会被decltype取得

应用场景: auto除了不能在定义类时使用, 还有一种不关心具体类型的目的在range-based for的场景尤其明显, 在; decltype则没有义类的使用限制, decltype会尤其关心具体...

展开 ∨

作者回复: 总结的挺好。

**yelin**

2020-05-27

1. 类型推导用起来方便, 代码里都用auto的话, 那个感觉应该和python是一样的吧, 所以不知道有没有好的解决方案, 我在python里的习惯是在命名加前缀, 还有就是注释了吧, 如果是协作开发的模块, auto我一般也就是用来内部遍历/range-based for之类的场景。请教下老师, 还有没有更好的办法。

展开 ∨

作者回复: 如果是用惯了Python等动态语言, 可能会对auto很适应。

但毕竟C++是强类型语言, 如果不小心推导出了你不需要的类型就会很麻烦, 所以为了“给人看”, 对于不是那么能够一眼看出来的类型, 最好还是注释说明一下。



湟水鱼儿

2020-05-25

公司编码不支持auto，只有自己私下里用一用，过一过瘾

作者回复: 唉，这个也是没办法的事情，以前我用的环境是gcc4.4，C++11支持不完整，非常痛苦。



汪zZ

2020-05-23

看auto的时候，省略了一下，觉得我这个初学者应该知道它存在就可以了，结果发现真的，有它更开心。

展开 ∨

作者回复: 这就是C++的哲学：自由，你怎么做都可以，总可以找到自己喜欢的风格。



silverhawk

2020-05-23

我说一个这种自动推导的隐藏代价吧，不是C++里面，C#里面的Var，有一次遇到一个Var res = func ()，这个func()返回一个IEnumerable，但是这个var就掩盖了这个IEnumerable究竟是List，还是其他什么，实际上背后有个stream的实现，之后程序中多次出现foreach res，其实就是多次遍历了这个stream，凭空增加了overhead。如果能够在写的时候显示定义，可能会想的更清楚，写出性能更高的

展开 ∨

作者回复: 对，对于某些很重要的类型，用auto后最好用注释说明它是个什么，后续该怎么用，否则会导致后面的代码比较难懂。



禾桃

2020-05-21

```
int x;
```

```
auto * y = &x; // auto 被推倒成int, y的类型是int*
```

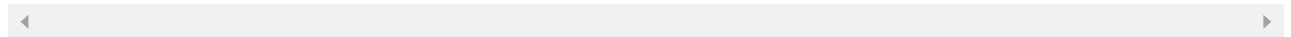

`auto z = &x; // auto 被推倒成int *, z的类型也是int*`

请问是这样吗? ...

展开 ▾

作者回复: 理解的很对。

auto还是比较智能的, 会自动推导出正确的类型, 注意它一定是值类型, 不会是引用。



💬 1



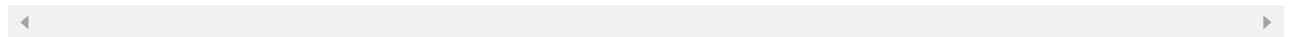
Zivon

2020-05-21

auto 和 decltype 虽然很方便, 但用多了也确实会“隐藏”真正的类型, 增加阅读时的理解难度, 确实有这方面的影响, 再看自己过去写的代码, auto会减慢阅读速度, 但IDE能提供一些辅助会好些。

auto 和 decltype 的区别。decltype能实现精确推导, auto一定不会推导出引用类型。 ...
展开 ▾

作者回复: 说的很好, auto配合适当的注释就比较完美了。



💬



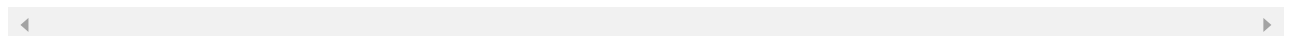
Luke

2020-05-21

C++14 auto用于推到函数返回值, 个人觉得是鸡肋, 增加代码阅读难度, 本来忘了返回值类型, 看一眼函数头就行了, 现在还得去看函数体具体的返回值类型到底是什么。这是为了迎合python程序员转C++么 😏

作者回复: C++会给我们多种选择, 我们也可以选择不用, 这个特性在写泛型函数的时候会很方便。

另外auto还可以用在返回值类型后置的用法, 有的特别的模板函数就真的只能这么写了。



💬



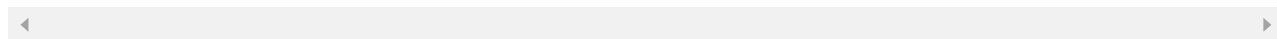
中年男子

2020-05-20

auto, decltype 用起来很方便,
说来惭愧, 以前一直很烦写函数指针的声明, 每次不得不写的时候都得google 一下,
自从用了decltype 再也不用担心了

auto 自动类型推导, 我最多的应用场景就是用来声明 stl的迭代器, 能少敲键盘, range...
展开 ∨

作者回复: auto和decltype用得少体会不出优越性, 只有写得多了, 在比较复杂的泛型场景下, 就会发现很多时候必须得用这两个关键字。

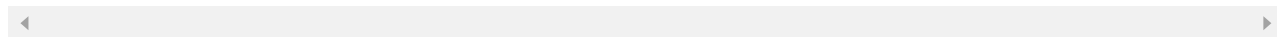


lckfa李钊

2020-05-19

我倒是觉得auto可以多用啊, 隐藏的真正类型完全可以使用vscode的cpp插件或者ide工具直接查看到, 不算大的缺点。如果说C++11让我最舒服的地方就是auto和using了。

作者回复: 我也比较同意, auto和using能很大程度改善代码。



范闲

2020-05-19

1. 大量使用auto和decltype确实会有这种问题。所以产量定义和初始化的时候原类型定义还是不错的.auto其实循环展开上用比较合理。decltype在类定义里使用, 不传递到外部。
- 2.auto和decltype其实更多是语法糖的效果。实际类型确定都在编译期。

展开 ∨

作者回复:

1.auto还可以用在复杂的模板类定义的时候, 比如容器的迭代器。

2 decltype可不能单纯地认为是语法糖, 它是编译期计算, 在泛型编程和模板元编程的时候非常有用, 像nullptr的类型, 就是用了decltype。



